VIRUS ANALYSIS

THE ROAD LESS TRUVELLED: W32/TRUVEL

Peter Ferrie
Microsoft, USA

Everything old is new again – at least for some virus writers.

By the addition of a relocation table, *Vista* executables can be configured to use a dynamic image base. That essentially turns them into executable DLLs. Now a virus has come along that has made a 'breakthrough' by infecting these executables – at least it would be a breakthrough if it weren't for the fact that relocatable executables have been supported since *Windows 2000* (ASLR in 1999!), and we have seen plenty of viruses that can infect DLLs. What's more, applications can have different image bases even without a relocation table, which from the virus's point of view amounts to the same thing. There is no need for a virus to carry absolute addresses – the alternative is a technique called 'relative addressing'.

LOCK AND LOAD

The virus, which we call W32/Truvel, begins by saving all registers and flags using the 'pusha' and 'pushf' instructions, as well as saving the ebp register explicitly (perhaps the virus author thought that the 'pusha' instruction was not sufficient). Then the virus determines its load address. This can be done simply by using a call→pop sequence, but the virus author seems to have wanted to make it more complicated. In this case, the load address is determined by calling a routine that sets up a structured exception handler, then intentionally causes an exception.

The handler receives control and retrieves the pointer to the context structure. It retrieves the original esp register value from the context structure, then fetches the return address from the stack and uses it to calculate the delta offset. The offset is stored in the ebp register within the context structure. Then the handler adjusts the eip register value in the context structure in order to skip the instruction that caused the exception, and returns control to the operating system to resume execution.

Interestingly, the handler contains an instruction to retrieve the base address of the Process Environment Block, but does nothing further with it. It is unclear what purpose this might have served in an exception handler. The first version of the virus also contains a check for the presence of a debugger by examining the 'BeingDebugged' flag in the Process Environment Block, but there is no branch instruction to take action if the flag is set – perhaps it was removed while debugging, and the virus author forgot to restore it. In the second variant of the virus the sequence has been removed completely.

SUCH HOSTILITY

Upon returning from the exception handler, the virus checks for the presence of a debugger by examining the 'BeingDebugged' flag in the Process Environment Block. If a debugger is detected, then the virus branches intentionally to an invalid address (which is the value of the eflags register), and the process terminates.

CRASH AND BURN

If no debugger is detected, the virus saves two image base values on the stack: the image base value from the Process Environment Block and the kernel32.dll image base value which it retrieves from the InLoadOrderModuleList structure. This leads to a problem, but only in the most unlikely circumstances, such as a bad memory layout in an emulator. Part of the problem is that if the kernel32.dll image base does not contain the right signatures (i.e. beginning with 'MZ', and with the lfanew field pointing to the PE header), then the virus attempts to clean up and run the host. The other part of the problem is that, at that point, no API addresses have been retrieved, so the cleanup will probably cause the application to crash.

In case the addresses saved during replication happen to match, the virus attempts to free two memory blocks that it has not yet allocated. This may not cause a crash, but another problem is caused by the fact that the two image base values that were saved onto the stack are not removed prior to the virus attempting to restore the registers and flags – which results in register corruption. However, even that might not be enough to cause a crash. The fatal blow comes in the form of the host entrypoint not having been adjusted according to the image base value, so the virus always branches to an invalid memory address and crashes.

Another bug exists in the code that attempts to locate the GetProcAddress() API. The virus loops through all of the APIs until GetProcAddress() is found. However, if for some reason the function is not found and the loop exits, the code continues its execution at the same location as that which is reached if the function is found. The result is that the virus resolves to an address which will likely point to an invalid memory address and cause a crash.

PROTECT AND SERVE

The virus calls VirtualProtect() to write-enable its code. This is the result of an anti-heuristic effect which will be explained below. If the call to VirtualProtect() fails for some reason, then, as above, the virus branches to the cleanup routine and crashes.

At this point, the virus removes the image base values from the stack, and adjusts the host entrypoint according to the image base value. Then comes some code of great silliness:

The virus wants to retrieve the addresses of some functions from kernel32.dll. While it is a simple matter to construct one relative pointer to the list of names and one relative pointer to the location at which to store the addresses, the virus writer chose another method. The virus carries a table of pairs of absolute addresses. One half of the pair points within the virus code to the name of the function to retrieve from kernel32.dll, while the other half points within the virus code to the location at which to store the retrieved address. Each of the addresses must be adjusted individually according to the delta offset, in order to locate the appropriate data. If any of the functions cannot be resolved, then the virus branches to the cleanup routine and attempts to free two memory blocks that it has not yet allocated. The list of functions includes entries that the virus does not even use.

LOSING MY MEMORY

The virus calls a function twice to allocate two blocks of memory for itself. However, after each call comes a check for failure. If the first allocation fails, then the virus branches to the cleanup routine and attempts to free the second block which it has not yet allocated.

If the allocations succeed, then the virus searches in the current directory for all files whose suffix is '.exe'. For each file that is found, the virus opens it and reads some data into one of the memory blocks. The virus checks for the 'MZ' signature, and the second variant includes some bounds checking on the lfanew field value prior to checking for the PE signature. The problem is that the bounds checking is incorrect.

Instead of checking whether the lfanew field value plus the size of the signature is not greater than the size of the block, the virus attempts to check only if the lfanew field value is less than the size of the block – and it even gets that wrong. The virus checks that the lfanew field value is not greater than the size of the block. This allows for an lfanew value that is exactly equal to the size of the block – also known as an off-by-one bug.

The problem is compounded by the fact that no further bounds checking is performed, leading to the assumption that if the PE header signature is within the block, then the entire PE header and the section table must be within the block.

EVUL IS AS EVUL DOES

The infection marker for the virus is a section named 'Evul', which is the name of the virus author. If no such section exists, then the virus simply appends one, without regard to the possible overflow of the block or the overwriting of the data in the first section. The virus then seeks the end of the file and calculates a new size according to the FileAlignment field value. If the file size was not aligned before, then the virus attempts to write enough data to align it. However, the stack is the source of the data to write, and if the amount of data to write is large enough, then it will fail. This result is not checked.

The virus calculates an aligned SizeOfRawData value for the original last section. If the value was not aligned already, then the virus replaces the old value with the new one, and applies the difference to the SizeOfImage value. This is another bug, since the SizeOfImage value comes from the sum of the VirtualSize values, not the sum of the SizeOfRawData values.

BACK AND FILL

The rest of the data for the new section are filled in at this point. The virtual address is calculated by aligning the VirtualSize of the previous section. The section characteristics specify a section that is read-only and executable. In the past, it was common for viruses to make the last section writable when they infected a file. It became such a common technique that some anti-virus programs still use it as a rule for performing more thorough scans of files. As a result, the absence of the writable bit can help some viruses to hide, at least for a while.

Next, the virus zeroes the LoadConfig and BoundImport data directory entries in the PE header. This has the effect of disabling the Safe Exception Handling, since the entries are located inside the LoadConfig data.

Finally, the virus writes itself to the file, updates the entrypoint to point to the new section, and writes the new PE header to the file. Then the virus searches for another file to infect.

The virus has no intentional payload, however its many bugs are sufficient to produce some surprises – it's amazing that the virus replicates at all.